5

The Maintenance Process

"Modelling is a tool for coping with problems of largeness"

DeMarco ([79], p 42)

This chapter aims to

1. Discuss the importance of process models.
2. Explain the weaknesses of traditional life-cycle models with respect to maintenance.
3. Identify ways of accommodating the evolutionary tendency of software within traditional software life-cycle models.
4. Study examples of maintenance process models.
5. Compare and contrast different types of maintenance process model.
6. Discuss the strengths and weaknesses of each maintenance process model.
7. Look at the issue of process maturity.

5.1 Introduction

We have looked at different types of change, why and when they might be carried out, and the context in which this will be done. This chapter will look at the change process by looking in detail at different ways in which the process of maintenance has been modelled. In order to study maintenance process models effectively, they need to be seen in the context of traditional life-cycle models. A comparison of traditional and maintenance models helps to highlight the differences between software development and software maintenance and shows why there is a need for a 'maintenance-conscious' process model.

5.2 Definitions

Life-cycle – the cyclic series of changes undergone by an entity from inception to 'death'. In terms of software, the life-cycle is the series of recognised stages through which a software product cycles during its development and use.

Model – the representation of an entity or phenomenon.

Process – the progress or course taken, methods of operation, a series of actions taken to effect a change.

"A specific activity or action performed by a human being or machine during a software project"

Basili & Abd-El-Hafiz ([15], p.3)

Process model – the representation of the progress or course taken – i.e. the model of the process.

Software maintenance process – the series of actions taken to effect change during maintenance.

5.3 The Software Production Process

The software production process encompasses the whole activity from the initial idea to the final withdrawal of the system (Figure 5.1).

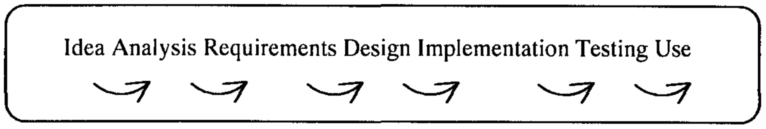
Idea Analysis Requirements Design Implementation Testing Use

Figure 5.1 Stages in the evolution of a software system

A process, being the series of discrete actions taken to effect a change, is distinct from the life-cycle which defines the order in which these actions are carried out.

The software life-cycle starts with an idea, then goes through the stages of feasibility study, analysis, design, implementation, testing, release, operation and use. The software evolves through changes which lead to, or spring from, new ideas that start the cycle off again. A familiar example is that of the metamorphosis of an insect (Figure 5.2). The adult insect lays the egg from which the pupa emerges which becomes the chrysalis from which the adult emerges which lays the egg... and so on.

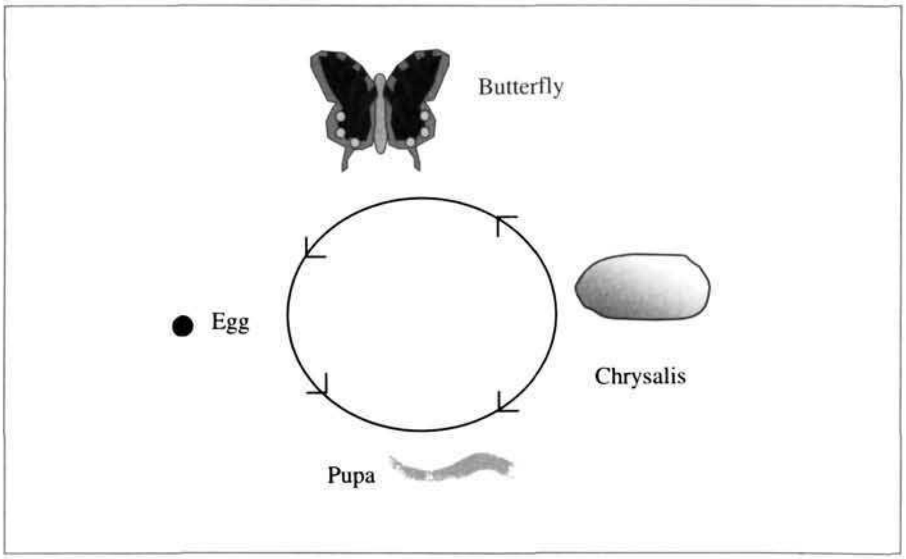

Figure 5.2 The metamorphosis of an insect

Mini Case Study – The Analysis of Patient Questionnaires at the ACME Health Clinic

The person whose job it was to analyse patient questionnaires became disillusioned with the tediousness of sifting through sheets of paper and collating responses. She had the idea of computerising the process and thereby automating it. The objectives were to speed the process up and to allow the computer to deal with the tedium of adding and collating, removing the time-consuming need for double-checking of calculations done by hand, thus making the process more reliable.

The technician had to ask the following:

a) Would it be feasible?

b) Could the job be done with available equipment and resources?

The answers being yes, the 'system' progressed to the next stage - a detailed appraisal of exactly what was needed. Once this was decided, the system was designed and implemented. Once implemented, the system was tested to the point where it was deemed reliable enough to use for real.

In this instance, one person creating a system for her own use progressed from the original idea to the system in use in a few hours. Interestingly, but not surprisingly, as soon as it became known that the system had been computerised, many requests for further data analysis came in. Not a single one of these could be accommodated. The person concerned had automated a specific set of analyses, with no consideration for further expansion.

We are familiar with the life-cycle as a cyclic series of stages (Figure 5.3) but what exactly comprises these stages?

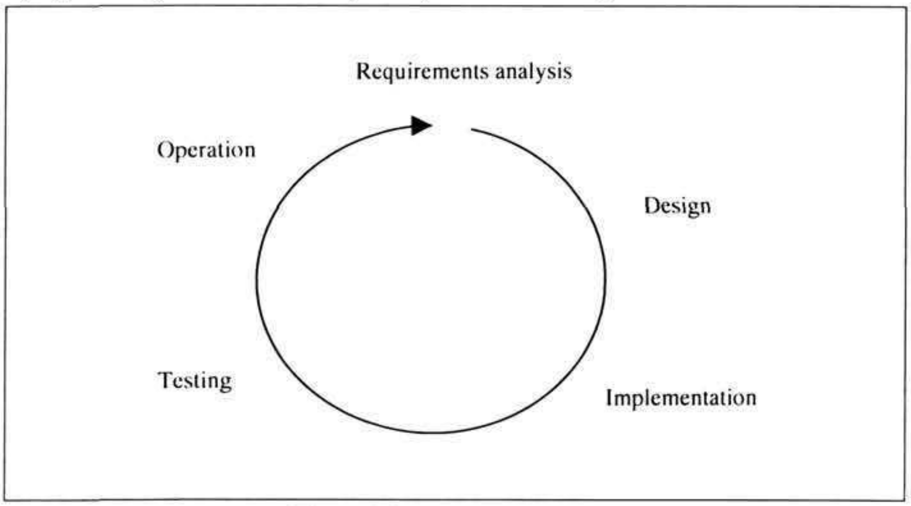

Figure 5.3 The generic life cycle

Design, for example, can be a pile of documents, a set of diagrams, ideas in someone's head, drawings on a blackboard. How do we deal with this plethora of information and make sense of it? We need a means of representing it in a way that hides the confusing detail and allows us to understand the major concepts. We need a model.

A familiar example is a map. A map provides an abstract representation of an area, and a very useful one. Given the real thing, the area, it is very difficult to know where anything is or how to find one's way about. Given an abstract and manageable representation, a map, the task becomes much easier.

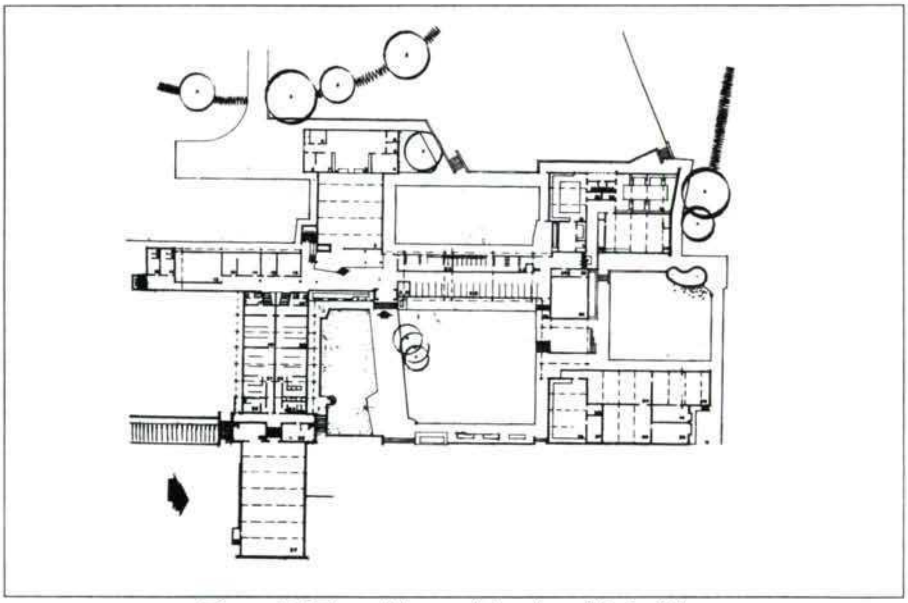

Figure 5.4 The architectural drawing of the building

Similarly, an architect's drawings (Figure 5.4) represent the information needed to construct a building. Such a representation gives more than just the idea of the shape or look of the building (Figure 5.5), it contains the information needed to enable safe construction of the building.

In software terms the model is the abstract representation of the software production process, the series of changes through which a software product evolves from initial idea to the system in use. For software maintenance, it is the representation of those parts of the process specifically pertaining to the evolution of the software.

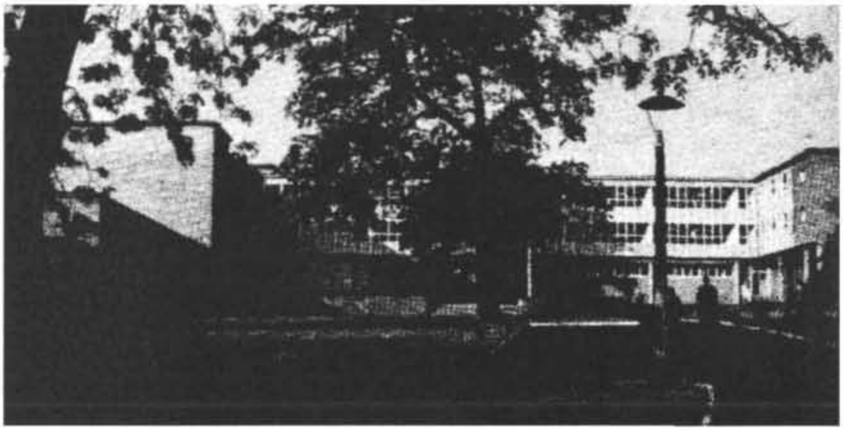

Figure 5.5 The finished building

A process model gives an abstract representation of a way in which to build software. Many process models have been described and we will look at a number of such models.

The term process implies a single series of phases. Life-cycle implies cycling through this series of phases repeatedly. However, you will find that some texts use the term software process as an alternative to software life-cycle. In this case, the term process as opposed to life-cycle is being used to give a different emphasis rather than implying a series versus a cyclic repetition of a series. The pivotal points of the software life-cycle in this case are the products themselves and software process shifts the emphasis to the processes by which the products are developed.

Exercise 5.1 Define the terms process, life-cycle and model.

Exercise 5.2 Explain the differences between a software life-cycle and a software process.

5.4 Critical Appraisal of Traditional Process Models

The history and evolution of life-cycle models is closely tied to the history and evolution of computing itself. As with other areas (the creation of programming languages, system design and so on) there was an evolutionary path from the *ad hoc* to the structured.

In the days when computer system development was a case of one person developing a system for his or her own use, there were no major problems with *ad hoc* and hit or miss methods. Such methods, in fact, are integral to the learning process in any field. As the general body of knowledge and experience increases, better understanding results and better methods can be developed.

Other factors influence the move from the *ad hoc* to the structured: risk factors, for example; safety considerations and the ramifications of failure when a large customer base has to be considered. Thus need, knowledge and experience lead to better structured and better understood models.

However, there is a specific aspect to be considered when looking at the historical development of models for maintenance. The evolution of models went in parallel with the evolution of software engineering and computer science in general. It must be remembered that the level of awareness of software maintenance-related issues was low until relatively recently. Software maintenance itself as a field of study is new compared to software development. The process and life-cycle models have evolved in an environment of high awareness of software development issues as opposed to maintenance issues and, as such, are development models.

There are very many software process and life-cycle models and, of these, many have a variety of permutations. In this section we will look at three which are representative of the area of process models in general: code-and-fix, waterfall and spiral, representing respectively the old, the well established and the new. An outline of these is given below to provide a framework for subsequent discussion of maintenance process models. The details of the traditional models are extensively covered in other texts [255, 274].

5.4.1 Code-and-Fix Model

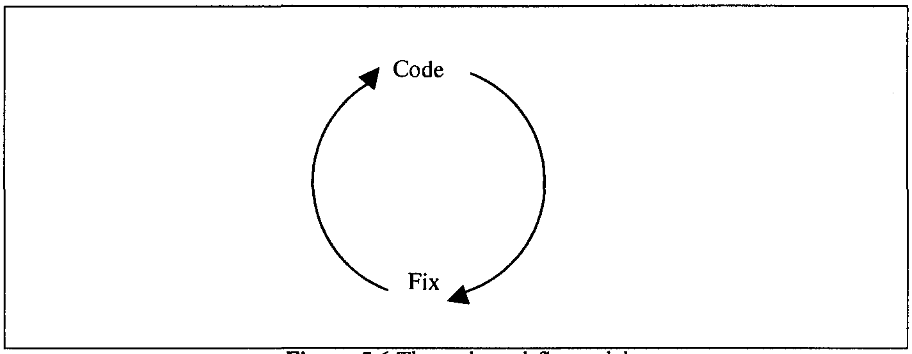

Figure 5.6 The code-and-fix model

This is *ad hoc* and not well defined. It is a simple two-phase model (Figure 5.6). The first phase is to write code. The next phase is to 'fix' it. Fixing in this context may be error correction or addition of further functionality. Using this model, code soon becomes unfixable and unenhanceable. There is no room in this model for analysis, design or any aspect of the development process to be carried out in a structured or detailed way. There is no room to think through a fix or to think of the future ramifications, and thus errors increase and the code becomes less maintainable and harder to enhance. On the face of it, this model has nothing to recommend it. Why then consider it at all? The reason is that despite the problems, the model is still used, the reason being that the world of software development often dictates the use of the code-and-fix model. If a correction or an enhancement must be done very quickly, in a couple of hours say, there is no time for detailed analysis, feasibility studies or redesign. The code must be fixed. The major problems of this scenario are usually overcome by subsuming code-and-fix within a larger, more detailed model. This idea is explored further in the discussion of other models.

The major problem with the code-and-fix model is its rigidity. In a sense, it makes no allowance for change. Although it perhaps does not assume the software to be correct from the off - it does have a fix stage as well as a code stage - it makes no provision for alteration and repair. The first stage is to code. All the other stages through which a software system must go (analysis, specification, design, testing) are all bundled together either into the fix stage or mixed up with the coding. This lack of properly defined stages leads to a lack of anticipation of problems.

Ripple effects, for example, will go unnoticed until they cause problems, at which stage further fixes may have become unviable or impossible. There is no acknowledgement in the model that one route may be better or less costly than another. In other, more structured models, explicit provision is made for the following of a particular route, for example a lesser risk route or a less expensive route. Because there is no such provision in the code-and-fix model, code structure and maintainability will inevitably deteriorate. Recognition of the problems of *ad hoc* software development and maintenance led to the creation of better structured models.

5.4.2 Waterfall Model

The traditional waterfall model gives a high-level view of the software life-cycle. At its most basic it is effectively the tried and tested problem-solving paradigm:

- Decide what to do
- Decide how to do it
- Do it
- Test it
- Use it.

The phases in the waterfall model are represented as a cascade. The outputs from one phase become the inputs to the next. The processes comprising each phase are also defined and may be carried out in parallel (Figure 5.7).

Many variations on this model are used in different situations but the underlying philosophy in each is the same. It is a series of stages where the work of each stage is 'signed off' and development then proceeds to the following phase. The overall process is document driven. The outputs from each stage that are required to keep the process moving are largely in the form of documents.

The main problem with the original waterfall model lay in its sequential nature, highlighted by later refinements which adapted it to contain feedback loops. There was recognition in this of the ever-increasing cost of correcting errors. An error in the requirements stage, for example, is far more costly to correct at a late stage in the cycle and more costly than a design error.

Nonetheless, the model still fails to capture the evolutionary nature of the software. The model allows for errors in the specification stage, for example, to be corrected at later stages via feedback loops, the aim being to catch and correct errors at as early a stage as possible. However, this still assumes that at some point a stage can be considered complete and correct, which is unrealistic. Changes - in specification for example - will occur at later stages in the life-cycle, not through errors necessarily but because the software itself is evolutionary.

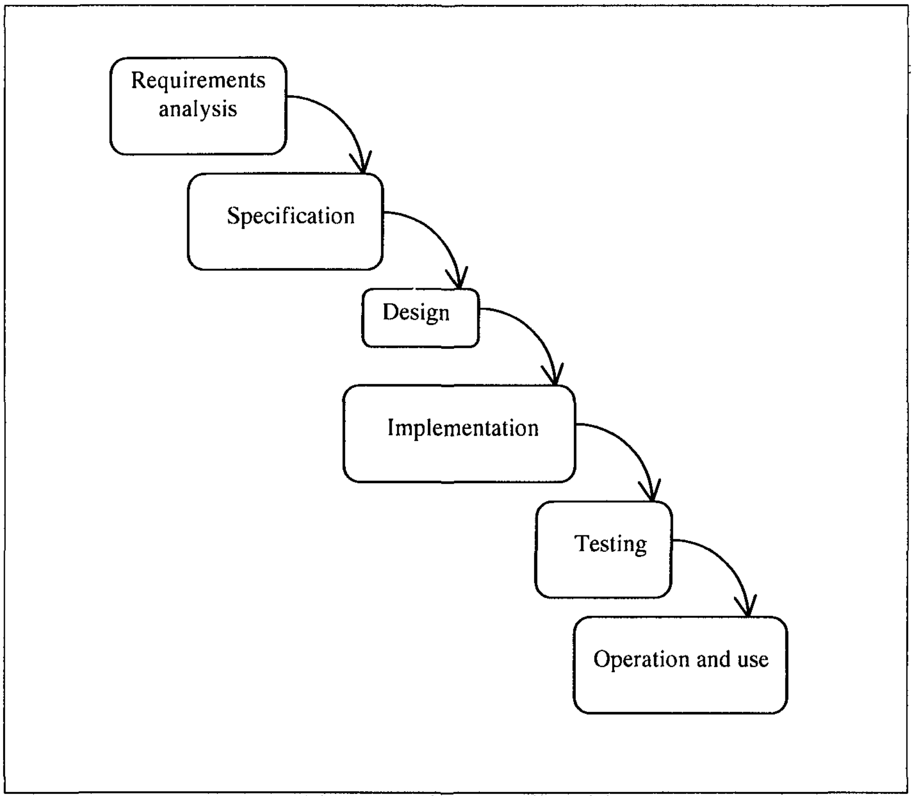

Figure 5.7 The waterfall model

A specification may be correct at a particular point in time but the system being specified is a model of some part of the world - complex air traffic control perhaps, or simple analysis of questionnaire answers. A system models an aspect of reality which is subject to change. Systems become incorrect not always through error or oversight but because we live in an ever-changing world and it is this evolutionary aspect of software systems that the waterfall model fails to capture.

More recently developed models take a less simplistic view of the life-cycle and try to do more to accommodate the complexities.

5.4.3 Spiral Model

The phases in this model are defined cyclically. The basis of the spiral model is a four-stage representation through which the development process spirals. At each level

- objectives, constraints and alternatives are identified,
- alternatives are evaluated, risks are identified and resolved,
- the next level of product is developed and verified,
- the next phases are planned.

The focus is the identification of problems and the classification of these into different levels of risk, the aim being to eliminate high-risk problems before they threaten the software operation or cost.

A basic difference between this and the waterfall model is that it is risk driven. It is the level of risk attached to a particular stage which drives the development process. The four stages are represented as quadrants on a Cartesian diagram with the spiral line indicating the production process (Figure 5.8).

One of the advantages of this model is that it can be used as a framework to accommodate other models. The spiral model offers great advantage in its flexibility, particularly its ability to accommodate other life-cycle models in such a way as to maximise their good features and minimise their bad ones. It can accommodate, in a structured way, a mix of models where this is appropriate to a particular situation. For example, where a modification is called for quickly, the risks of using the code-and-fix scenario can be evaluated and, if code-and-fix is used, the potential problems can be addressed immediately by the appropriate procedures being built into the next phase.

A problem with the spiral model is a difficulty in matching it to the requirements for audit and accountability which are sometimes imposed upon a maintenance or development team. The constraints of audit may be incompatible with following the model; for example, a very tight deadline may not allow sufficient time for full risk evaluation. This may well be an indication that the constraints imposed in terms of audit and accountability are less than optimal.

The fact that the model is risk driven and relies heavily on risk assessment is also a problem area. In breaking down a problem and specifying it in detail, there is always a temptation to 'do the easy bits first' and leave the difficult bits until last. The spiral model requires that the high-risk areas are tackled first and in detail. Although 'difficult' does not always equate to 'high risk' it often does. A team inexperienced in risk assessment may run into problems.

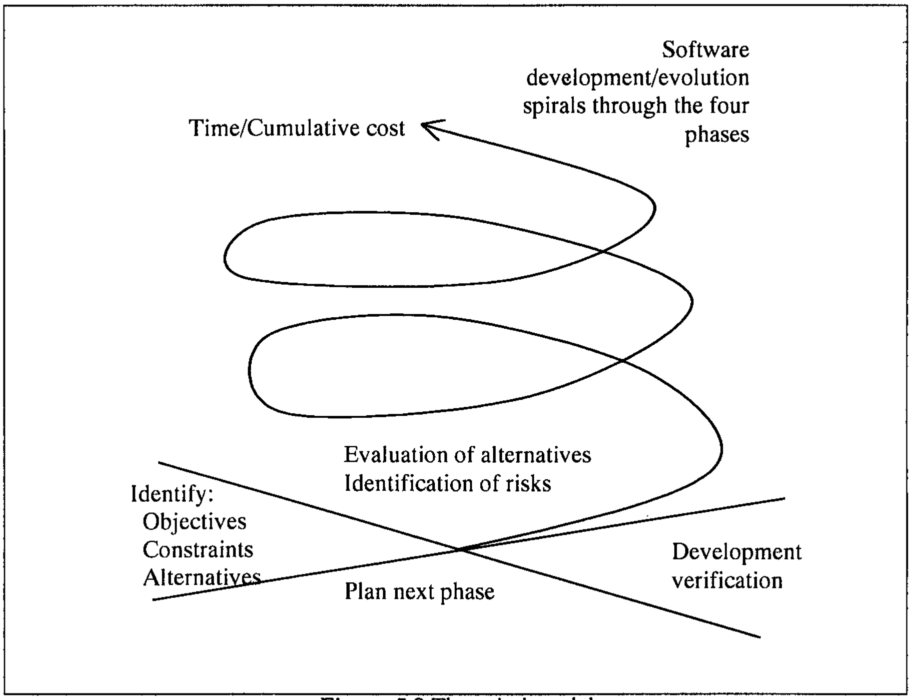

Figure 5.8 The spiral model

Exercise 5.3 Investigate the ways in which the basic waterfall model has been enhanced and modified. Describe these modifications and explain why they were made.

5.5 Maintenance Process Models

The need for maintenance-conscious models has been recognised for some time but the current situation is that maintenance models are neither so well developed nor so well understood as models for software development.

In the early days, problems with system development were overwhelming and it is not surprising that the evolutionary nature of software that is at the heart of maintenance was to an extent ignored [26]. To attempt to take account of future changes in systems, prior to good understanding of the development process, was akin to asking for the incorporation of a crystal ball into the model. However, our understanding of the maintenance process, just like our understanding of the development process, moved on and maintenance process and life-cycle models emerged.

Expanding on the example given in chapter 1, let us consider the addition of a room to a building. When the house was built to its original design, rooms A and B were built side by side. Some years later, a third room is needed. Had this need been perceived originally, three smaller rooms would have been built (Figure 5.9).

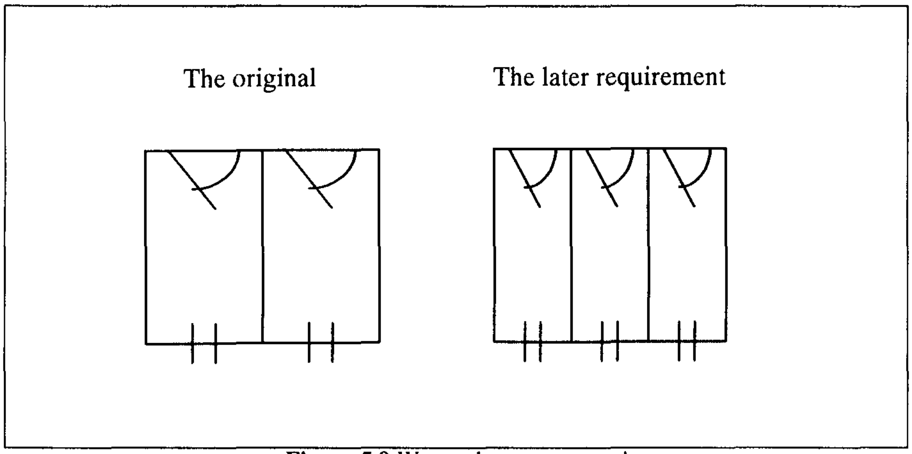

Figure 5.9 We need an extra room!

At the time of the original building, there was no need for a third room. But, after the building had been in use for some time, a need for a third room emerged. The addition of the third room to the existing building is a very different proposition from constructing the third room in the first place.

This is directly analogous to the addition of new requirements to a software system. Planning to build three rooms from the start is relatively easy, as is initial development of a particular functionality. Deciding to add the third room prior to commencement of building work is a little harder and requires alteration of plans. This equates to the case where there is a change in requirements subsequent to designing a piece of software but prior to implementing it. The addition of the extra room after the building has been completed and is in use is a very different matter, as is modification to software which is in use.

- The wall between rooms A and B must be knocked down.

  Software interfaces between different components may have to be altered.

- This is a building in use - the problem of creating and removing a pile of rubble must be addressed. The resultant dust with which the original building site could have coped may now pose a major threat to sensitive equipment. At the time of original building, rubbish chutes and skips would have been on site.

  There is far less leeway to allow for the introduction of errors and ripple effects in a piece of software which must be released quickly to a large customer base. During initial development, there was a specific and resourced testing phase. Reintroduction of modified software may be subject to tight time deadlines and resource constraints.

- Adding the third room may well require people and materials to travel through, and thus affect, parts of the building they would not have had to access originally. The work will impact differently upon the environment upon which it is carried out. The effects of adding the third room as opposed to building it in the first place will cause disruption at a different level, to a different group of people and in different places. All this needs to be assessed and addressed.

  Similarly, a modification to a large and complex software system has the potential to affect parts of the software from which it could have been kept completely separate had it been added originally.

- The physical effect on the building itself will be different. Is the wall between A and B a load-bearing wall? If so, there will be a need for a supporting joist. Had the original plans catered for a third room, there would have been no supporting joist across the middle of a room in this way. It would have been unnecessary, a design flaw in fact, had it appeared in the original, and yet in the conversion it is an absolute necessity

  The software will have to be modified to cater for the addition of the new functionality. Suppose that the new functionality calls for data to be held in memory in a large table. It may be that the existing system does not allow the creation of such a structure because of memory constraints. Data structures in other parts of the system may not have left enough room for a large table. If it is not feasible to make extensive alteration to the rest of the data structures, then something other than a large table must be used. This something else, a linked list perhaps, may seem wholly inappropriate. And yet, the demands of the maintenance environment insist upon it.

- Does the wall contain central heating pipes, wiring ducts, network cables or anything else which may have to be taken into account prior to its being demolished?

  Likewise, are there hidden dependencies within the software modules which are to be modified? In theory, these will all be documented. In practice, buildings tend to be better documented than software systems.

It is all too easy to assume that an enhancement to an existing software system can be tackled in exactly the same way as adding that feature from the start. This misconception may be due to the malleable nature of software. It is not as obvious with software, as it is with a building, that adding something later is a very different case from adding it in the first place.

It is this concept, that maintenance is a very different matter from initial development, that a maintenance-conscious model must encompass.

One can go only so far along the road towards predicting a future need for an extra room. And yet traditional engineering methods allow us to add new rooms to buildings without having to demolish the building or make it unsafe. In software engineering, there is a great deal of demolition work going on simply because we cannot add the 'extra room' safely. Predicting every future need is not possible and attempting to do so is very costly. We can, however, do more to encompass the genuine needs of maintenance within the models with which we work.

An obvious example is documentation. Engineers in other fields would not dream of neglecting, or working without, documentation. If proper documentation did not exist, road workers would cut off mains services almost every time they dug a hole in a city street. Yet software engineers often have to rely on their own investigations to discover dependencies between software modules or to discover potential ripple effects because the original system and subsequent changes are not documented.

It is important to recognise the differences between new development and maintenance but it is also important to recognise the similarities. In the building analogy; it is the same skills and expertise that are required to build the new wall whether constructing it in the new building or adding it later. What will make the difference is whether the work is being done on a building site or in a building in use.

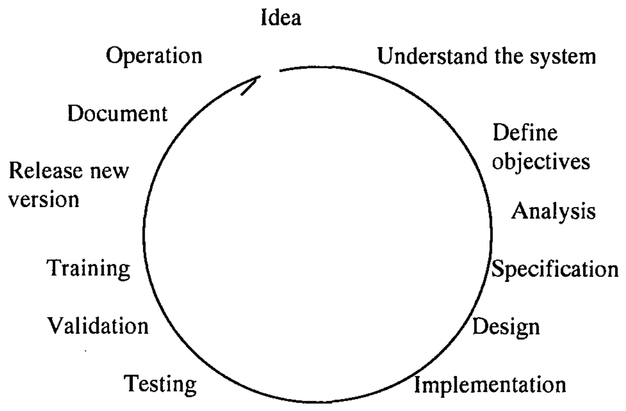

Figure 5.10 The 'maintenance conscious' life cycle

The generic stages in a maintenance-conscious model (Figure 5.10) compared with the traditional development model appear similar on the surface but within the stages there are great differences in emphasis and procedure. There is more effort required and very different emphases on the early stages, and conversely less effort required in the later stages, of the maintenance model as opposed to the development model (Figure 5.11).

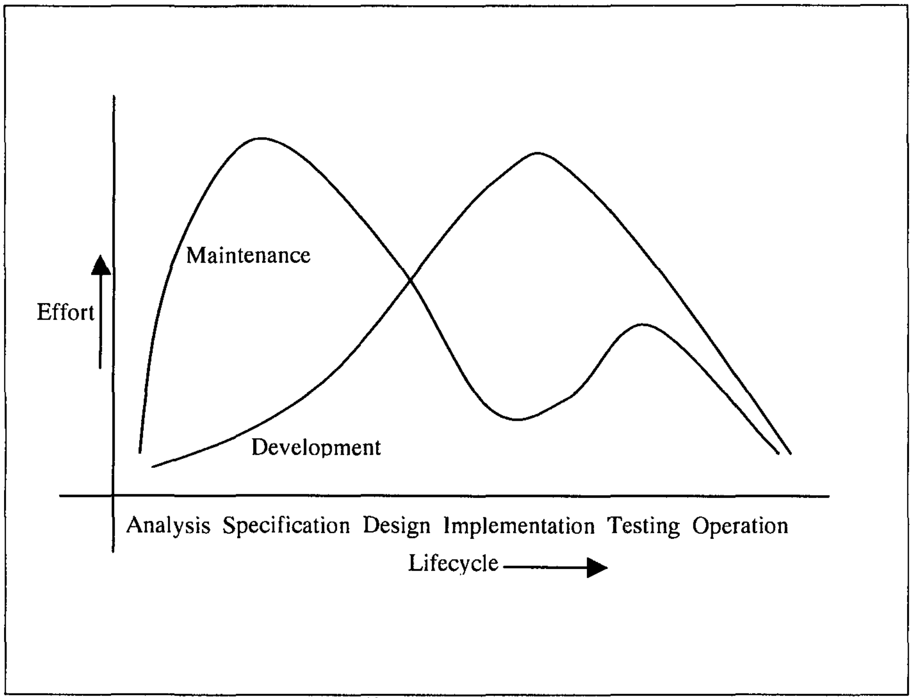

Figure 5.11 Effort needed in the different stages

Consider the building example again. Buildings are built for a particular purpose but often change use during their lifetime. A private house is converted to a shop. A stately home is converted to offices. The fact that the private house was not originally built with provision for a shop window or a counter does not mean that the house must be entirely demolished. Appropriate skills are brought to bear and the required conversions carried out. If the hopeful shopkeepers were to say to the builder 'If you were building this window into a new building whose design had allowed for it, it would take you x person-hours and you would use these specific materials and spend z pounds. Here are z pounds, build it this way,' the builder would turn down the job and wonder why these people wanted to insist on dictating details of something they appeared to know nothing about. And yet, much software maintenance is carried out this way. Had an original specification allowed for a particular functionality, it might have taken as little as five minutes to implement. 'Please deliver the modified software in five minutes!' Is it any surprise that software collapses under conversion to a greater extent than buildings do?

The essence of the problems at the heart of all the traditional models is in their failure to capture the evolutionary nature of software. A model is needed which recognises the requirement to build maintainability into the system. Once again, there are many different models and we will look only at a representative sample of four of them.

5.5.1 Quick-Fix Model

This is basically an *ad hoc* approach to maintaining software (Figure 5.12). It is a 'firefighting' approach, waiting for the problem to occur and then trying to fix it as quickly as possible, hence the name.

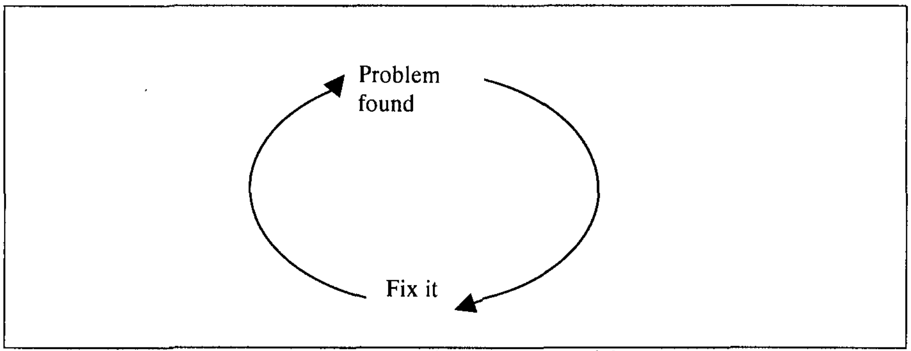

Figure 5.12 The quick-fix model

In this model, fixes would be done without detailed analysis of the long-term effects, for example ripple effects through the software or effects on code structure. There would be little if any documentation. It is easy to see how the model emerged historically, but it cannot be dismissed as a purely historical curiosity because, like the code-and-fix model, it is still used.

What are the advantages of such a model and why is it still used? In the appropriate environment it can work perfectly well. If for example a system is developed and maintained by a single person, he or she can come to learn the system well enough to be able to manage without detailed documentation, to be able to make instinctive judgements about how and how not to implement change. The job gets done quickly and cheaply.

However, such an environment is not the norm and we must consider the use of this model in the more usual setting of a commercial operation with a large customer base. Why does anyone in such a setting still allow the use of an unreliable model like the quick-fix? It is largely through the pressure of deadlines and resources. If customers are demanding the correction of an error, for example, they may not be willing to wait for the organisation to go through detailed and time-consuming stages of risk analysis. The organisation may run a higher risk in keeping its customers waiting than it runs in going for the quickest fix. But what of the long-term problems? If an organisation relies on quick-fix alone, it will run into difficult and very expensive problems, thus losing any advantage it gained from using the quick-fix model in the first place.

The strategy to adopt is to incorporate the techniques of quick-fix into another, more sophisticated model. In this way any change hurried through because of outside pressures will generate a recognised need for preventive maintenance which will repair any damage done.

By and large, people are well aware of the limitations of this model. Nonetheless, it often reflects only too well the real world business environment in which they work. Distinction must be made between short-term and long-term upgrades. If a user finds a bug in a commercial word processor, for example, it would be unrealistic to expect a whole new upgrade immediately Often, a company will release a quick fix as a temporary measure. The real solution will be implemented, along with other corrections and enhancements, as a major upgrade at a later date.

5.5.1.1 Case Study – Storage of Chronological Clinical Data

When the ACME Health Clinic system was originally developed, it catered only for a single recording per patient for things such as blood pressure, weight, medication and so on. This was because of a misunderstanding during requirements analysis which did not come to light until the system was in use. In fact, the system needed to store chronological series of recordings. At that stage, the need for storage of chronological data was immediate. The maintenance programmer assigned to the task drew up a mental model of data held in small arrays to allow speedy retrieval and proceeded to implement the change. This quick-fix method identified the need

- for the arrays,
- to amend the data structures to allow for linking of the chronological data,
- for a small restructuring program to modify the existing data.

There was no update of documentation, no documentation of the changes other than a few in-code comments and no in-depth analysis.

Because this was done speedily as a quick fix, problems such as array overflow were not considered. In fact, once enough information was stored, data was going to 'drop off the end' of the arrays and disappear. This would lead to data corruption in that the chronological links would be broken and missing links would appear in the middle of the data chains (Figure 5.13).

This was noticed while another enhancement was being tested, the potential seriousness of the problem was recognised and the race was on to solve it before the clinic stored sufficient data to cause the problem. This imposed yet another tight deadline and the fastest fix had to be found. The 'best' solution, a radical restructuring of the data and procedures for data retrieval and storage, was recognised, but could not be implemented because of the time restriction. Another quick fix had to be found. The only solution was to 'catch' data overflowing the temporary array and store it in the patient file. This meant that chronological links were maintained, but the data was being stored in the patient's file without being explicitly saved by the clinician. This not only led to less well-structured code, documentation further out of date and a situation even harder to retrieve, but was also in contravention of an original requirement regarding permanent saving of data.

This ACME Health Clinic case study highlights the difficulties of the quick-fix model. Ripple effects that should have been obvious were missed. These forced the adoption of a further fix which was known to be wrong. The resources that had to be diverted into this emergency repair lessened the likelihood of time being devoted to documentation update, thus decreasing the chances of the error being successfully retrieved.

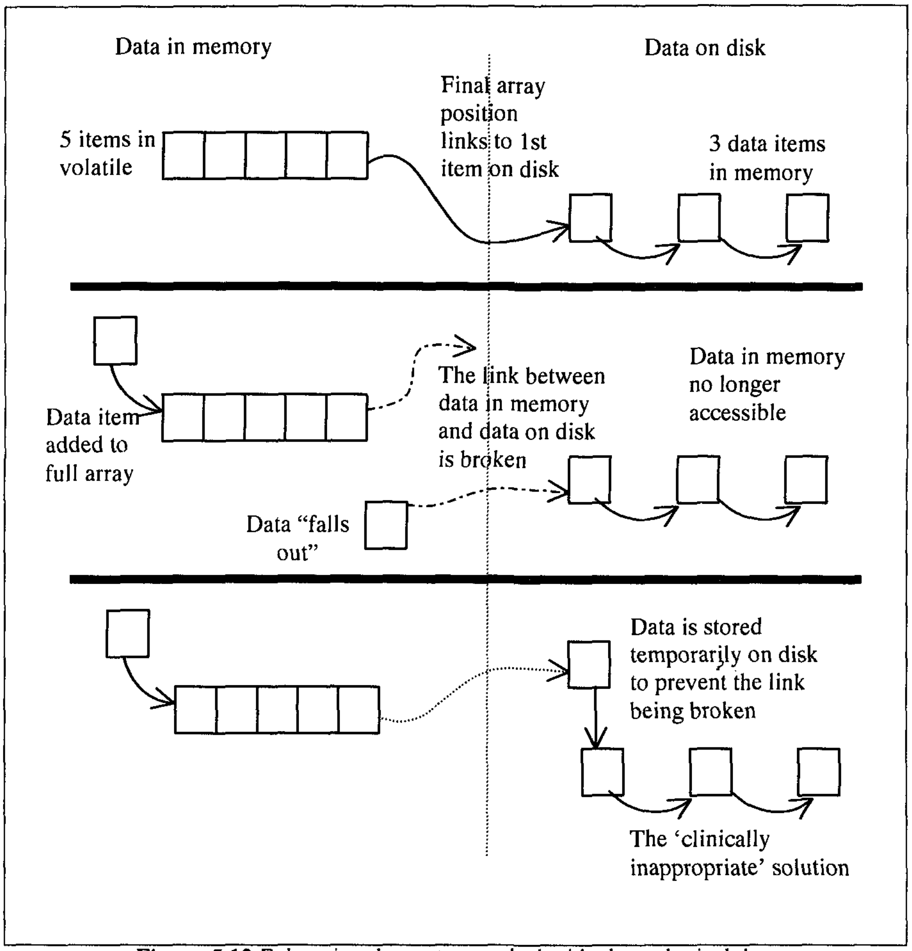
Data in memory
Data on disk
Final array position links to 1st item on disk
5 items in volatile
3 data items in memory
Data item added to full array
The link between data in memory and data on disk is broken
Data in memory no longer accessible
Data "falls out"
Data is stored temporarily on disk to prevent the link being broken
The 'clinically inappropriate' solution

Figure 5.13 Enhancing the system to deal with chronological data

The underlying basis of the problem is that the quick-fix model does not 'understand' the maintenance process. The problems experienced were not hard to predict and the 'advantage' gained by the original quick fix was soon lost.

Many models have subsequently been developed which look at, and try to understand, the maintenance process from many different viewpoints. A representative selection of these is given below.

5.5.2 *Boehm's Model*

In 1983 Boehm [36] proposed a model for the maintenance process based upon economic models and principles. Economic models are nothing new. Economic decisions are a major driving force behind many processes and Boehm's thesis was that economic models and principles could not only improve productivity in maintenance but also help understanding of the process.

Boehm represents the maintenance process as a closed loop cycle (Figure 5.14). He theorises that it is the stage where management decisions are made that drives the process. In this stage, a set of approved changes is determined by applying particular strategies and cost-benefit evaluations to a set of proposed changes. The approved changes are accompanied by their own budgets which will largely determine the extent and type of resource expended.

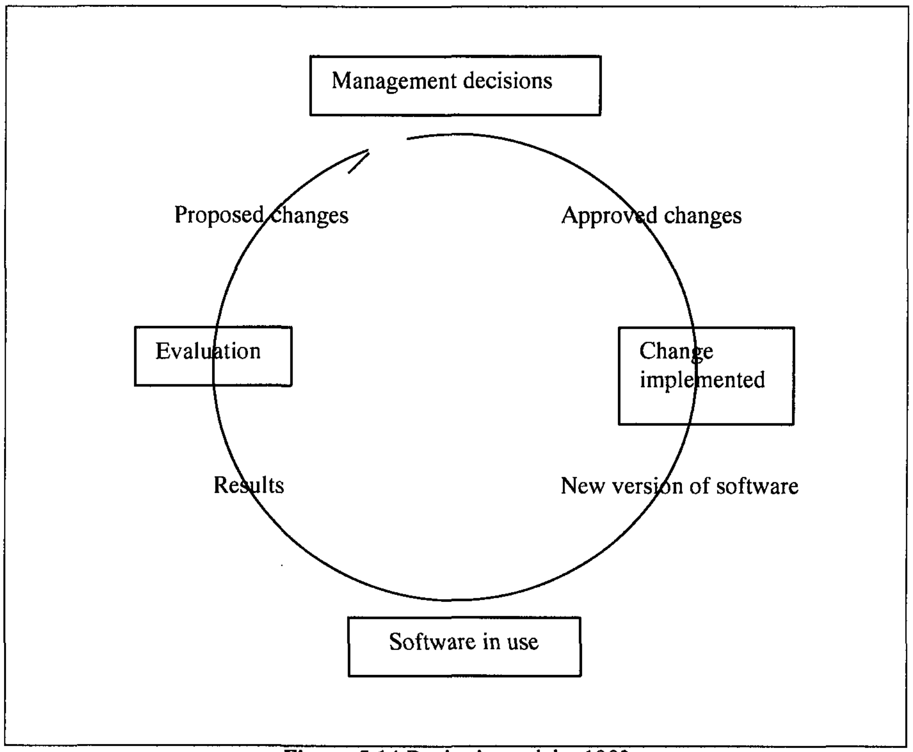

Figure 5.14 Boehm's model – 1983

The survey by Leintz and Swanson [176] (Figure 5.15) showed that almost half maintenance effort was devoted to non-discretionary maintenance activities.

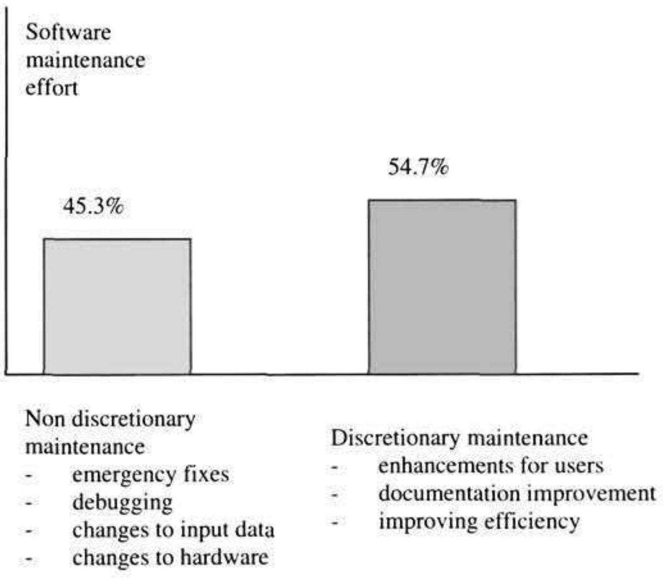

Figure 5.15 Results of Lientz and Swansons' survey – 1978

In terms of the production function – the economic relationship between the inputs to a process and its benefits – this reflects the typical three-segment graph of:

- *Investment:* This is a phase of low input of resource and low benefit. This correlates to a newly released software product which has a high requirement for emergency fixes and mandatory enhancements.
- *High payoff:* An organisation sees increasing benefit from the software product and the initial problems are ironed out. This is a phase during which resource is put into user enhancements and improvements in documentation and efficiency. Cumulative benefit to the organisation increases quickly during this phase.
- *Diminishing returns:* Beyond a certain point, the rate of increase of cumulative benefit slows. The product has reached its peak of

usefulness. The product has reached the stage where radical change becomes less and less cost effective.

Boehm [36] sees the maintenance manager's task as one of balancing the pursuit of the objectives of maintenance against the constraints imposed by the environment in which maintenance work is carried out.

Thus, the maintenance process is driven by the maintenance manager's decisions which are based on the balancing of objectives against constraints.

In the example of the problems with the ACME Health Clinic system, this approach to maintenance would have recognised that the quick-fix approach adopted was not appropriate. Had a quick fix been essential, it would have been a temporary holding measure which would have allowed the system to continue running without radical and ill-thought-out changes. These would have been assessed as part of the overall strategy and would have allowed a progression towards the real solution instead of the inevitable path away from it.

5.5.3 Osborne's Model

Another approach is that proposed by Osborne [210]. The difference between this model and the others described here is that it deals directly with the reality of the maintenance environment. Other models tend to assume some facet of an ideal situation - the existence of full documentation, for example. Osborne's model makes allowance for how things are rather than how we would like them to be.

The maintenance model is treated as continuous iterations of the software life-cycle with, at each stage, provision made for maintainability to be built in. If good maintenance features already exist, for example full and formal specification or complete documentation, all well and good, but if not, allowance is made for them to be built in.

The stages in the maintenance life-cycle are shown in Figure 5.16 and include recognition of the steps where iterative loops will often occur.

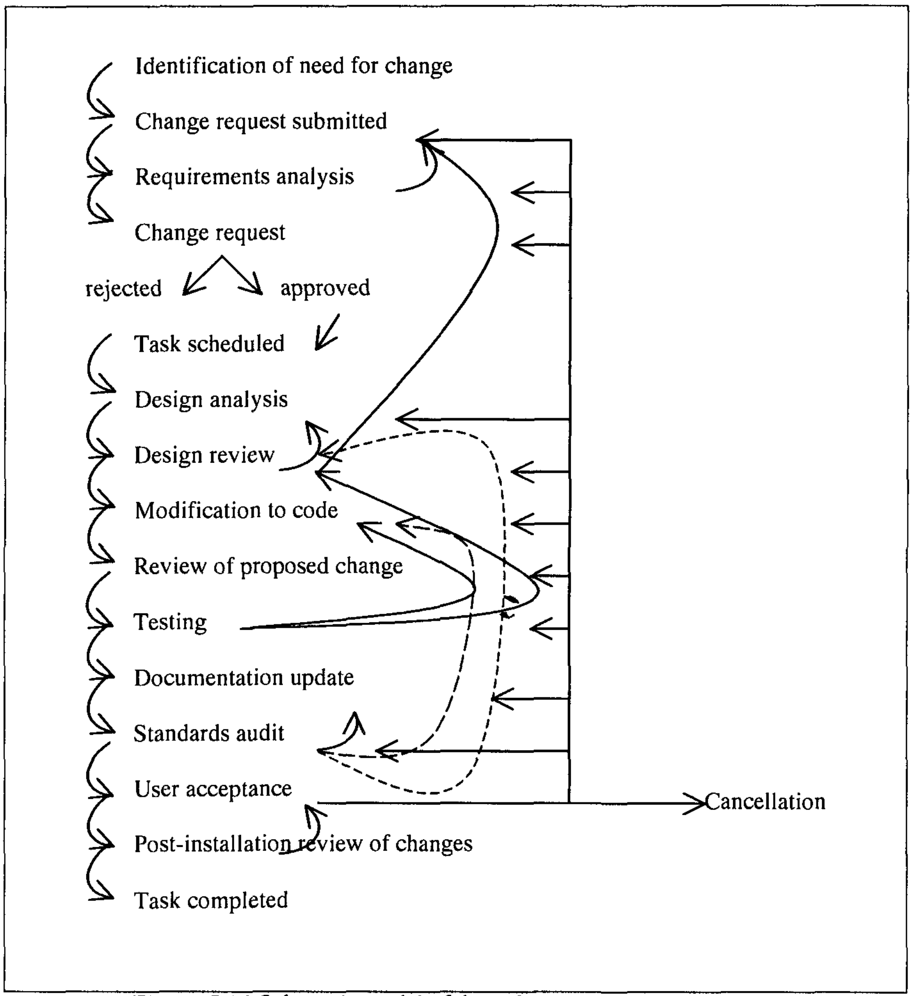

Figure 5.16 Osborne's model of the software maintenance process

Osborne hypothesises that many technical problems which arise during maintenance are due to inadequate management communications and control, and recommends a strategy that includes:

- the inclusion of maintenance requirements in the change specification;
- a software quality assurance program which establishes quality assurance requirements;

- a means of verifying that maintenance goals have been met;
- performance review to provide feedback to managers.

5.5.4 *Iterative Enhancement Model*

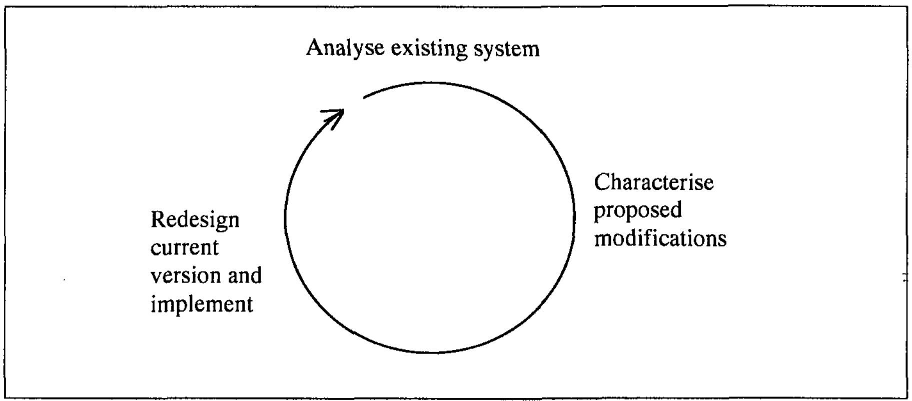
Analyse existing system
Characterise proposed modifications
Redesign current version and implement

Figure 5.17 The three stages of iterative enhancement

This model has been proposed based on the tenet that the implementation of changes to a software system throughout its lifetime is an iterative process and involves enhancing such a system in an iterative way. It is similar to the evolutionary development paradigm during pre-installation.

Originally proposed as a development model but well suited to maintenance, the motivation for this was the environment where requirements were not fully understood and a full system could not be built.

Adapted for maintenance, the model assumes complete documentation as it relies on modification of this as the starting point for each iteration. The model is effectively a three-stage cycle (Figure 5.17):

- Analysis.
- Characterisation of proposed modifications.
- Redesign and implementation.

The existing documentation for each stage (requirements, design, coding, testing and analysis) is modified starting with the highest-level document affected by the proposed changes. These modifications are propagated through the set of documents and the system redesigned.

The model explicitly supports reuse (see chapter 8) and also accommodates other models, for example the quick-fix model.

The pressures of the maintenance environment often dictate that a quick solution is found but, as we have seen, the use of the 'quickest' solution can lead to more problems than it solves. As with the previous model, iterative enhancement lends itself to the assimilation of other models within it and can thus incorporate a quick fix in its own more structured environment. A quick fix may be carried out, problem areas identified, and the next iteration would specifically address them.

The problems with the iterative enhancement model stem from assumptions made about the existence of full documentation and the ability of the maintenance team to analyse the existing product in full. Whereas wider use of structured maintenance models will lead to a culture where documentation tends to be kept up to date and complete, the current situation is that this is not often the case.

5.5.5 Reuse-Oriented Model

This model is based on the principle that maintenance could be viewed as an activity involving the reuse of existing program components. The concept of reuse is considered in more detail in chapter 8. The reuse model described by Basili [16] has four main steps:

- Identification of the parts of the old system that are candidates for reuse,
- Understanding these system parts,
- Modification of the old system parts appropriate to the new requirements,
- Integration of the modified parts into the new system.

A detailed framework is required for the classification of components and the possible modifications. With the full reuse model (Figure 5.18) the starting point may be any phase of the life-cycle – the requirements, the design, the code or the test data – unlike other models. For example, in the quick-fix model, the starting point is always the code.

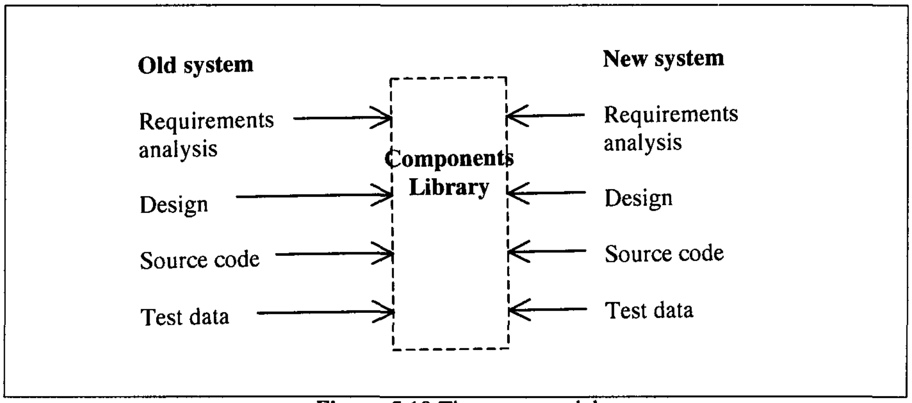

Figure 5.18 The reuse model

Exercise 5.4 Compare and contrast Osborne's maintenance process model with the other maintenance process models dealt with in this chapter.

Exercise 5.5 Describe how the ACME Health Clinic system might have been more effectively modified. Assume the same tight deadlines but investigate the incorporation of the quick-fix model into another, more structured model.

5.6 When to Make a Change

So far discussion has been about the introduction of change into a system without considering whether or not that change should be made at all. In other words, the ways in which different models approach the implementation of change has been considered but without addressing the important question of how to decide when a change should be made. It cannot simply be assumed that everyone involved with a system, from the developers to the users, can throw their ideas into the arena and automatically have them implemented. That would lead to chaos.

Not all changes are feasible. A change may be desirable but too expensive. There has to be a means of deciding when to implement a change. Ways of doing this, e.g. via a Change Control Board, are explored in detail in chapter 11.

Exercise 5.6 What was the last software project you worked on? Was it a commercial project, an undergraduate project or a personal project? Write a critical appraisal of the life-cycle model to which you worked. Was it well structured or *ad hoc?* Would you work to a different model if you were to start this project again?

Exercise 5.7 You are the IT manager in charge of a large library software system which fails unexpectedly one Monday morning. How would you go about the process of solving this problem

1. if it is imperative that it is up and running within two hours?
2. if the library is able to function adequately for several days without its software system?

5.7 Process Maturity

We have looked at how processes are modelled, but a vital issue is of course, how they are used.

Knowledge of the theory does not lead automatically to effective use in practice. Many undergraduate software engineering programmes include a group project where students work together on a large software project, to mimic the commercial environment. Processes will have been learnt in other parts of the course. If the application of these to the group work is *ad hoc* and not controlled, the results of such projects will be unpredictable. Outcomes will depend upon chance, individual flair of team members and will by and large be random. Well-organised projects, in contrast, should allow all groups to use effectively the processes they have learnt in their theoretical courses.

A similar situation holds in a commercial software house. Some companies carry on successful operations for a long time reliant on a few very good programmers. If they do not put resources into building the maturity of the processes themselves, there will come a point where the operation cannot continue. The burden will become far too great for the few people carrying it. They will leave and the whole operation will collapse.

Organisations need a means by which to assess the maturity and effectiveness of their processes.

5.7.1 *Capability Maturity Model® for Software*

The Software Engineering Institute (SEI) developed a capability maturity model for software [217]. Using this, the maturity of processes can be assessed. Five levels are defined:

1) **Initial.** The software process is *ad hoc*. Few processes are defined. Success depends on individual flair of team members.

2) **Repeatable.** Basic processes are established, tracking cost, scheduling, and functionality. Successes can be repeated on projects with similar applications.

3) **Defined.** Processes are documented and standardised. There exists within the organisation standard processes for developing and maintaining software. All projects use a tailored and approved version of the standard process.

4) **Managed.** Detailed measures are collected, both of the process and the quality of the product. Quantitative understanding and control is achieved.

5) **Optimising.** Quantitative feedback is evaluated and used from the processes and from the piloting of innovative ideas and technologies. This enables continuous process improvement.

The SEI's model is not the only one in use, but is widely referenced and other models [63] tend to be closely cross-referenced with it. The benefits accruing from software process improvement based upon the SEI's model have been studied and documented [128, 161].

5.7.2 *Software Experience Bases*

The idea of an organisation continually improving through sharing experience is a concept that can counter the vulnerability inherent in having experience concentrated in a few skilled employees.

Knowledge can be formalised into guidelines and models, or may be embodied in the skills of the personnel involved. The latter is as much an asset as a company's software systems, built using such knowledge, but harder to turn into an asset that can effectively be shared and retained.

Organisations have created systems to support knowledge and experience sharing e.g. knowledge and experience databases, with varying degrees of success. Conradi *et al* [67] suggest that software experience base is a more useful term than database, to avoid inappropriate comparison with the traditional database management systems. They propose four factors required for successful implementation of a software experience base:

1. Cultural change – people must become comfortable with sharing knowledge and using others' knowledge and experience, in order that the software experience base is active and used.
2. Stability – an unstable business environment will not be conducive to the development of a culture or a system for knowledge and experience sharing.
3. Business value – in order for any system to be used and useful in today's business world, it must provide a demonstrable payback.
4. Incremental implementation – implementing a software experience base in small increments is of use in keeping the process close to the users and, with effective feedback, prevents the software experience base becoming a remote and irrelevant entity.

5.8 Summary

The key points that have been covered in this chapter are:

- The software life cycle is the cyclic series of phases through which a software system goes during its development and use. A process is a single series of such phases.
- A process model abstracts the confusing plethora of detail from the process of software evolution and allows us to understand it.
- Traditional life-cycle models fail to take account of the evolutionary nature of software systems.
- There are major differences between new development and maintenance although they have many specific phases in common. Maintenance-conscious models can be built from traditional models by catering for the evolutionary tendency of the software.
- There are many different maintenance models. Three representative ones are quick-fix which is an *ad hoc*, fire-fighting approach; iterative enhancement which is based on the iterative nature of

change to a system; and reuse-oriented which sees maintenance as an activity involving the reuse of program components. The most pragmatic approach is given by Osborne's model.

- Models differ in bias: some are oriented towards economic aspects, some towards products and some towards processes.
- All models have strengths and weaknesses. No one model is appropriate in all situations and often a combination of models is the best solution.
- By improving the maturity of their software processes, software developers and maintainers move from as-hoc, ill-defined processes where success is largely a matter of chance, to a situation of predictable and repeatable project outcomes and continual improvement.

This part of the book has put maintenance activities into context and looked at how the maintenance process is modelled. The next stage is to look at what actually happens during maintenance.